North Carolina Journal of
Mathematics and Statistics
Volume 4, Pages 29–55 (Accepted August 28, 2018, published September 24, 2018)
ISSN 2380-7539

The `booleantools` Package: An Open-Source Python Framework for Boolean Functions

Andrew Penland and Wesley Rogers

ABSTRACT. Boolean functions are crucial in the design of secure cryptographic algorithms. We introduce `booleantools` , an open-source Python package for the analysis and design of boolean functions. As an example of the software's functionality, we show how it can be used to find geometric information about the space of all boolean functions on 5 variables.

1. Introduction

Boolean functions are one of the mathematical building blocks used in the construction of algorithms for cryptography and coding theory. As just one example, boolean functions are used in the *Secure Hash Standard* developed by the National Institute of Standards and Technology and required under federal law for securing classified data (National Institute of Standards and Technology, 2015). In this paper, we present `booleantools` , a Python package designed to facilitate the computational analysis of boolean functions.

We let $\mathbb{F}_2 = \{0, 1\}$ denote the finite field with two elements, equipped with the usual operations: addition modulo 2, denoted by $\oplus$, and multiplication modulo 2, denoted by juxtaposition. We write $\mathbb{F}_2^n$ for the n-dimensional vector space over $\mathbb{F}_2$. A *boolean function* is a map from $\mathbb{F}_2^n$ to $\mathbb{F}_2$. It is well-known that such a function can always be represented as a polynomial in n variables with coefficients in $\mathbb{F}_2$, and that is the representation we will typically use here. We will provide further mathematical background in Section 3.

There are several properties that are accepted as being necessary for cryptographic security of boolean functions, including *resiliency*, *nonlinearity*, lack of *linear structures*, high *algebraic degree*, etc. We give definitions of these properties in Section 3. We highly recommend Carlet (2010) for an in-depth overview of these topics.

The designer of cryptographically secure boolean functions faces many challenges. Brute force enumeration of all possible functions on n variables becomes infeasible very quickly as n grows. Another challenge is that the various necessary properties are in conflict. A result due to Siegenthaler (1984) shows that as the resiliency of an n-variable boolean function increases, the algebraic degree necessarily decreases. Hence, finding suitable boolean functions is a multi-objective optimization problem which requires sophisticated computation in conjunction with mathematical analysis.

The `booleantools` package offers useful functionality that is not directly available in any existing open source Python package. The Python packages `PyCrypto` (Litzenberger, 2018) and `pyca/cryptography` (`pyca/cryptography` Developers, 2018) offer high-level implementations of cryptographic algorithms for software developers, but they do not make boolean

2010 *Mathematics Subject Classification.* 12-04; 68N01.

Key words and phrases. coding theory; cryptography; computational algebra; `Python`; group theory; boolean functions; hamming distance.

functions or their properties easily accessible. Another package, `PolyBoRi`, also works with polynomial functions over $\mathbb{F}_2$. The `booleantools` package differs from `PolyBoRi` in two ways: `booleantools` is written entirely in Python, and `booleantools` has built-in support for many algebraic and computational manipulations of interest in cryptography and coding theory. While packages in `GAP` (GAP, 2018), `SageMath` (The Sage Developers, 2018), and other computer algebra systems could be used to do the same things that `booleantools` does, they are large multipurpose tools, with large code bases. Since `booleantools` is implemented in Python, it can be seamlessly integrated with other Python packages for machine learning (such as `Scikit-learn` (Pedregosa et al., 2011)) and evolutionary algorithms (such as `deap` (Fortin et al., 2012)), both of which are established techniques for search and analysis of boolean functions(see, for instance, Asthana et al. (2014) and Sadohara (2001)).

2. Technical Description

The `booleantools` package is a Python package that allows the user to create boolean functions using many different representations. Once the boolean function is created, `booleantools` provides built-in methods to analyze the function for various properties discussed in the literature. Additionally, `booleantools` provides support for actions of permutation groups on functions, and geometric tools such as Hamming and Hausdorff distance functions.

2.1. Installation and Requirements

The `booleantools` package is available on PyPi at

`https://pypi.python.org/pypi/booleantools`

and can be installed using the standard Python package manager `pip`, by typing

```
pip3 install booleantools
```

from a command line prompt.

The `booleantools` package requires any version of Python 3. For Python versions before Python 3.3, the user may need to first install `pip` by following the instructions available via `https://pip.pypa.io/en/stable/installing/` (The Python Packaging Authority, 2017).

The commands in this paper will be from `booleantools` version 0.4.2.

3. Background & Examples

In this section we discuss properties of boolean functions that are relevant for cryptographic research. For each property, we give the definition, as well as a code snippet demonstrating how `booleantools` can be used to compute this property. Our discussion of these properties follows Chapter 4 of Carlet (2010).

We write $[n]$ for the set $\{0, 1, \ldots, n-1\}$. We will use the vector notation $\mathbf{x}$ to indicate an element of a vector space $\mathbb{F}_2^n$, with the variable x_i representing the i'th coordinate of $\mathbf{x}$ for $i \in [n]$. A monomial is the product of the elements in some subset of $\{x_0, x_1, \ldots, x_{n-1}\}$.

It is well-known that any boolean function from $\mathbb{F}_2^n$ can be represented as a polynomial on the variables $x_0, x_1, \ldots, x_{n-1}$ using Lagrange interpolation (see Lidl and Niederreiter (1994)). This leads to writing boolean functions as a sum of monomials, and that is the representation we typically use for `booleantools` .

In some instances, it is helpful to identify a vector in $\mathbb{F}_2^n$ with its integer interpretation as a binary number, via the correspondence $\mathbf{x} \Leftrightarrow \sum_{k=0}^{n-1} x_k 2^k$. For instance, under this representation, the vector $[1,0,1,0]$ in $\mathbb{F}_2^n$ would correspond to $(1)2^0 + (0)2^1 + (1)2^2 + (0)2^3 = 6$. This facilitates the representation of an n-variable boolean functions via a *rule table*, a list of length $L = 2^n$ with indices taken from 0 to $L-1$, and the value at the k'th index given by the value of f on the boolean vector corresponding to k. As an example, the 4-variable boolean function $f(\mathbf{x})$ given by $f(\mathbf{x}) = x_0 x_1$ would have a rule table of:

$\mathbf{x}$ (integer form)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$f(\mathbf{x})$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Before implementing any of the code examples below, the user should import `booleantools` into their Python file or session. One approach would be:

```
from booleantools import *
```

For the remainder of the paper, we will use the functions g and h as examples, defined as follows.

$$\mathbf{x} = (x_0, x_1, ..., x_4)$$
$$g(\mathbf{x}) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0 x_1$$
$$h(\mathbf{x}) = x_0 x_1 \oplus x_2 x_3 \oplus x_4$$

This is easily translated to `booleantools` code.

```
>>> x = getX(5)
>>> g = BooleanFunction([[1],[2],[3],[4],[0,1]], 5)
>>> h = x[0]*x[1] + x[2]*x[3] + x[4]
```

A BooleanFunction object can be evaluated at a given point by treating the object as a function and passing in either the vector entries directly or a list of values.

Below are two different methods of evaluating the function g at a point.

```
>>>g(1,1,0,0,0)
>>>g([1,0,1,0,1])
```

```
0
0
```

The *Hamming weight* of an n-variable boolean function f is defined as the number of elements in $\mathbb{F}_2^n$ that f maps to 1. We write $\mathrm{wt}(f)$ for the Hamming weight of f. An n-variable boolean function f is *balanced* if $\mathrm{wt}(f) = 2^{n-1}$, i.e. if the number of elements that f maps to 1 is exactly the half the size of the domain. This is implemented in `booleantools` through the function `is_balanced`.

```
>>> g.is_balanced()
>>> h.is_balanced()
```

```
True
True
```

The *Hamming distance* between two n-variable boolean functions r and q is defined as

$$d(r, q) = \mathrm{wt}(r \oplus q)$$

and gives the number of elements of $\mathbb{F}_2^n$ for which r and q disagree. Hamming distance is implemented in `booleantools` as follows:

```
>>> g.hamming_distance(h)
```

```
16
```

The *dot product* on $\mathbb{F}_2^n$ is defined as $d(r, q) \pmod 2$ and denoted by $r \cdot q$.

The *Walsh transform* is a useful tool in the study of boolean functions. For an n-variable boolean function f, the *Walsh transform of f*, denoted by Wf, is a real-valued function defined for a vector $\mathbf{v} \in \mathbb{F}_2^n$ by:

$$(Wf)(v) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus x \cdot v}.$$

```
>>> g.walsh_transform()
```

```
[0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0, 0, -16,
    0, 0, 0, 0, 0, 0, 0, 16]
```

A boolean function f is called *k-th order correlation immune* if when we hold any k variables constant, the result (viewed as a function on $n - k$ variables) has the same proportion of 1's in the output as the original function. Our implementation for determining *k-order correlation immunity* is based on the Walsh transform, utilizing a criterion established by Xiao and Massey (1988), which says that a function f is k-correlation immune if and only if $Wf(\mathbf{v}) = 0$ whenever $\mathrm{wt}(\mathbf{v}) \leq k$.

```
>>> g.is_correlation_immune(k=2)
>>> h.is_correlation_immune(k=2)
```

```
True
False
```

A boolean function f is k-resilient if it is balanced and k'th-order correlation immune. We note that $g(\mathbf{x})$ is 2-resilient, and $h(\mathbf{x})$ is only 1-resilient.

```
>>> g.is_k_resilient(k=2)
>>> h.is_k_resilient(k=2)
```

```
True
False
```

If x_i is a variable such that flipping the value of x_i (i.e. replacing x_i with $x_i \bigoplus 1$) also changes the value of $f(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{F}_2^n$, then x_i is called a *linear structure* for the function f.

The method `linear_structures` returns the linear structures of a boolean function, as a set.

```
g.linear_structures()
```

```
{2, 3, 4}
```

The *algebraic degree* of a monomial is defined as the number of variables in the product; the degree of x_1 is 1, the degree of $x_0x_1x_2$ is 3, etc. The *algebraic degree* of an arbitrary boolean function is defined as the maximum of the algebraic degrees of its monomials; the algebraic degree of $x_1 \oplus x_2$ is 1,the algebraic degree of $x_1x_2 \oplus x_3$ is 2,etc.

A boolean function is *affine* if its algebraic degree is equal to 1. The *nonlinearity* of a boolean function f is defined as the minimum distance from f to any element in the space of affine functions. If f is linear, its nonlinearity is 0. We can use the `booleantools` functions `is_affine` and `nonlinearity` to determine these properties. Our implementation of the `nonlinearity` function also relies on the Walsh transform.

```
>>> g.nonlinearity()
>>> h.nonlinearity()
>>> g.is_affine()
>>> h.is_affine()
```

```
8
12
False
False
```

The Hamming distance leads us to further geometric considerations on the space of n-variable boolean functions. The development of the `booleantools` package arose from our interest in studying the space of boolean functions using ideas from geometry and group theory. We now briefly review a few necessary ideas in this vein.

If X is a set, a *permutation* on X is a bijective function from the set to itself. We write $\mathrm{Sym}(n)$ for the set of all permutations on $[n]$, known as the *symmetric group* on $[n]$. Let f be a boolean function, σ be a permutation in $\mathrm{Sym}(n)$, and x_i an indeterminate in the polynomial ring of $\mathbb{F}_2$. We define the function f^σ to be given by $f^\sigma(x_i) = f(x_{\sigma(i)})$, for $i \in [n]$ i.e. the coordinates are permuted before the boolean function is applied.

```
>>> perms = Sym(5)
>>> h.apply_permutation(perms[0])
```

```
BooleanFunction([[0, 1], [2, 3], [4]], 5)
```

The *diameter* of a set with respect to some distance function is the maximum distance obtained by any pair of points in the set. We can define a distance between two *sets* of boolean functions as well. If X and Y are two sets of n-variable boolean functions, the *Hausdorff distance* between X and Y is defined as:

$$d_H(X,Y) = \max\{\max_{x\in X}\min_{y\in Y} d(x,y), \max_{y\in Y}\min_{x\in X} d(x,y)\}.$$

Below we show how to obtain the Hausdorff distance between the equivalence classes of our example boolean functions:

```
>>> hausdorff_distance(g.get_orbit(), h.get_orbit())
```

```
12
```

It is worth noting that the nonlinearity of an n-variable boolean function f is exactly the Hausdorff distance from the singleton set $\{f\}$ to the set of all affine functions on n variables.

The deep relationships between algebraic structure, geometry, and group theory in coding theory and cryptography remain a subject of active research. We suggest the classic Conway and Sloane (2013) as a starting point for the interested reader.

3.1. Other Functions and Features

In addition to the functions given in the previous subsection, there are a large number of other functions which ease the ability to analyze particular classes of functions. We have included documentation for a list of them below. The most recent documentation is available on the package github page.

Sym(n)

Input: an integer (`int`) n
Returns: a list of all possible permutations as a list

```
>>>Sym(2)
```

```
[(0,1), (1, 0)]
```

getX(n)

Input: an integer (`int`) n
Output: a list of functions of the form $f(\mathbf{x}) = x_i$, for $0 \leq i < n$.

```
>>> getX(2)
```

```
[BooleanFunction([[0]], 1), BooleanFunction([[1]], 2)]
```

generate_function(rule_number, n)

Input: a rule number, which is an integer given by the base-2 encoding of the rule table.
Output: a boolean function on n variables with the specified rule number.

```
>>> generate_function(120, 3)
```

```
BooleanFunction([[1, 2], [0]], 3)
```

weight_k_vectors(k, nbits)

Input: k, the desired weight, and $nbits$, the number of bits
Output: a list containing all vectors in $\mathbb{F}_2^n$ with weight exactly equal to k

```
>>> weight_k_vectors(2, 3)
```

```
[[1, 1, 0], [1, 0, 1], [0, 1, 1]]
```

weight_k_or_less_vectors(k, nbits)

Input: k, the desired weight, and $nbits$, the number of bits
Output: a list containing all vectors in $\mathbb{F}_2^n$ with weight less than or equal to k

```
>>> weight_k_or_less_vectors(2, 3)
```

```
[[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 1]]
```

duplicate_free_list_polynomials(list_of_polys)

Input: A list of polynomials.
Output: The `duplicate_free_list_polynomials` function takes a list of polynomials, and returns the list with duplicates removed.

```
>>> duplicate_free_list_polynomials([BooleanFunction([[1], [1,2]], 3),
      BooleanFunction([[1,2], [1]], 3)])
```

```
[BooleanFunction([[1], [1, 2]], 3)]
```

orbit_polynomial(polynomial, permset)

Input: a polynomial (represented as a BooleanFunction object) and optionally a set of permutations

Output: the orbit of the polynomial under the permutation set

```
>>> orbit_polynomial(BooleanFunction([[1]], 2), Sym(2))
```

```
[BooleanFunction([[1]], 2), BooleanFunction([[0]], 2)]
```

orbit_polynomial_list(polynomial_list, permset)

Similar to `orbit_polynomial`, but for a list of polynomials.
Input: list of polynomials (with each polynomial represented as a list of monomial lists) and a set of permutations

Output: the orbit of the polynomials in `polynomial_list` under the set of all permutations in `permset`.

```
>>> orbit_polynomial_list([BooleanFunction([[1]], 2), BooleanFunction([[0]],
    2)], Sym(2))
```

```
[ BooleanFunction([[1]], 2), BooleanFunction([[0]], 2),
  BooleanFunction([[0]], 2), BooleanFunction([[1]], 2) ]
```

siegenthaler_combination(f1,f2,new_var)

Input: two n-variable boolean functions, f_1 and f_2, represented as `booleanfunction` objects
Output: A `booleanfunction` representing what we call the *Siegenthaler combination* of f_1 and f_2, both of which are boolean functions on n variables. This was introduced in Siegenthaler (1984), and is defined as

$$g(x) = x_n f_1(x) \oplus (1 \oplus x_n) f_2(x).$$

The Siegenthaler combination has the property that it increases the number of variables from n to $n+1$, while keeping the resiliency the same, and without introducing any additional linear structures.

```
>>> f1 = BooleanFunction([[1]], 2)
>>> f2 = BooleanFunction([[0], [0,1]], 2)
>>> nv = BooleanFunction([[2]], 3)
>>> siegenthaler_combination(f1, f2, nv)
```

```
BooleanFunction([[1, 2], [0], [0, 1], [0, 2], [0, 1, 2]], 3)
```

generate_all_seigenthaler_combinations(func_list,new_var)

Input: a list of `booleanfunction` objects
Output: a list giving all possible Siegenthaler combinations of the functions, without removing duplicates.

```
>>> f1 = BooleanFunction([[1]], 2)
>>> f2 = BooleanFunction([[0], [0,1]], 2)
>>> f3 = BooleanFunction([[2]],2)
>>> nv = BooleanFunction([[3]], 3)
>>> generate_all_siegenthaler_combinations([f1,f2,f3],nv)
```

```
[ BooleanFunction([[1]], 4),
  BooleanFunction([[1, 3], [0], [0, 1], [0, 3], [0, 1, 3]], 4),
  BooleanFunction([[1, 3], [2], [2, 3]], 4),
  BooleanFunction([[0, 3], [0, 1, 3], [1], [1, 3]], 4),
  BooleanFunction([[0], [0, 1]], 4),
  BooleanFunction([[0, 3], [0, 1, 3], [2], [2, 3]], 4),
  BooleanFunction([[2, 3], [1], [1, 3]], 4),
  BooleanFunction([[2, 3], [0], [0, 1], [0, 3], [0, 1, 3]], 4),
  BooleanFunction([[2]], 4)]
```

reduce_to_orbits(f_list, permset)

Input: a list of functions `f_list` and a set of permutations `permset`

Output: a list of representatives from each class, under the action of `permset` on `f_list`

```
>>> reduce_to_orbits([BooleanFunction([[0]], 2), BooleanFunction([[1]], 2)],
    Sym(2))
```

```
[BooleanFunction([[0]], 2)]
```

In addition to the methods for the analysis of boolean functions, there are several convenience functions.

Addition and multiplication of functions

The addition and multiplication of functions is fully supported, using standard Python notation for addition and multiplication. This is seen above, in the Python construction of function h.

BooleanFunction.tex_str(math_mode=False)

Output: a LaTeX representation of this function, along with proper math mode support if `math_mode` is set to True.

Example:

```
>>> print(g.tex_str())
>>> print(h.tex_str(math_mode=True))
```

```
x_{1} \oplus x_{2} \oplus x_{3} \oplus x_{4} \oplus x_{0}x_{1}
$x_{0}x_{1} \oplus x_{2}x_{3} \oplus x_{4}$
```

When rendered in LaTeX, we get the representations

$$g(\mathbf{x}) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0x_1$$
$$h(\mathbf{x}) = x_0x_1 \oplus x_2x_3 \oplus x_4$$

3.2. Source Code

The full source code of the `booleantools` package is available in Appendix B, and on GitHub at `https://github.com/MagicalAsh/BooleanFunctions`.

4. Example - Geometry of 2-Resilient Boolean Functions

As an example of the utility of the `booleantools` package, we demonstrate how we used it explore the geometry of the space of 2-resilient nonlinear boolean functions on 5 variables. We only consider nonlinear functions, since affine functions are known to be cryptographically insecure.

It should be noted that we are not the first to consider these functions. All 5-variable 2-resilient boolean functions were examined in Braeken et al. (2005), which determined cryptographic properties of all boolean functions on six variables or less.

Using the `booleantools` package, we were able to determine all nonlinear 2-resilient Boolean functions on five variables by exhaustive search. We then sorted the functions into their orbits under the symmetric group. The following table summarizes our findings for the orbits of the nonlinear 2-resilient boolean functions on five variables. The code we used is available in Appendix A, and took approximately 20 minutes to run on a personal computer running Debian 4.14 on a twenty core Intel Xeon. Additionally, after verifying the classes of 2-resilient Boolean functions, we calculated the diameter of each class, along with the linear structures of each class.

The code in Appendix A works by producing every possible quadratic polynomial on five variables, then sorting the resulting polynomials based on their resiliency. After processing all possible

TABLE 4.1. The class representatives, diameter, orbit size, and linear structures of 5-variable 2-resilient boolean functions

Orbit Representative	Linear Structures	Number in Orbit
$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0x_1$	x_2, x_3, x_4	10
$x_1 \oplus x_3 \oplus x_4 \oplus x_0x_2$	x_1, x_3, x_4	10
$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0x_1$	x_2, x_3, x_4	20
$x_2 \oplus x_4 \oplus x_0x_1 \oplus x_0x_3 \oplus x_1x_3$	x_2, x_4	10
$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_0x_1 \oplus x_0x_4 \oplus x_1x_4$	x_2, x_4	30
$x_1 \oplus x_3 \oplus x_4 \oplus x_0x_1 \oplus x_0x_2$	x_3, x_4	60
$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_0x_2 \oplus x_0x_1$	x_2, x_3	60
$x_1 \oplus x_3 \oplus x_4 \oplus x_0x_1 \oplus x_0x_2 \oplus x_1x_3 \oplus x_2x_3$	x_4	60

Orbit Number	Orbit Representative	Diameter of Orbit
1	$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0x_1$	12
2	$x_1 \oplus x_3 \oplus x_4 \oplus x_0x_2$	12
3	$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0x_1$	16
4	$x_2 \oplus x_4 \oplus x_0x_1 \oplus x_0x_3 \oplus x_1x_3$	12
5	$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_0x_1 \oplus x_0x_4 \oplus x_1x_4$	16
6	$x_1 \oplus x_3 \oplus x_4 \oplus x_0x_1 \oplus x_0x_2$	24
7	$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0x_1 \oplus x_1x_2$	20
8	$x_2 \oplus x_4 \oplus x_5 \oplus x_1x_2 \oplus x_1x_3 \oplus x_2x_4 \oplus x_3x_4$	24

quadratic functions, it then sorts the functions into their orbits under the action of the symmetric group, i.e. permutation of variables. The resulting output is a representative from each class of functions, output as a json file.

It may be executed from a command line as

```
python3 generate_classes.py 5
```

5. Conclusion, Future Work, and an Invitation

The reality is that any software package can be extended and improved. We intend to continue to develop `booleantools` , adding more support for cryptographic tests, methods for properties from coding theory, and support for group theory and group actions. We also intend to further optimize the existing methods for efficiency, as well as providing more support for multithreading and parallelization.

The code will be maintained at the second author's `GitHub` repository (at `https://github.com/MagicalAsh/BooleanFunctions`) and as a package on PyPi (`https://pypi.python.org/pypi/booleantools`).

We invite questions, suggestions and feature requests from interested parties. Those interested in contributing code or ideas for improvement are welcome to do so through a pull request at the GitHub repository.

Acknowledgments.

We would like to thank the Department of Mathematics and Computer Science at Western Carolina University for use of its computer laboratory, and Geoff Goehle for advising us on use of the lab. We would also like to thank John Asplund for a careful reading and many useful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 1524607.

References

Asthana, R., Verma, N., and Ratan, R. (2014). Generation of boolean functions using genetic algorithm for cryptographic applications. In *Advance Computing Conference (IACC), 2014 IEEE International*, pages 1361–1366. IEEE. http://ieeexplore.ieee.org/document/6779525/.

Braeken, A., Borissov, Y., Nikova, S., and Preneel, B. (2005). Classification of boolean functions of 6 variables or less with respect to some cryptographic properties. In *International Colloquium on Automata, Languages, and Programming*, pages 324–334. Springer. https://link.springer.com/chapter/10.1007/11523468_27.

Carlet, C. (2010). Boolean functions for cryptography and error correcting codes. *Boolean models and methods in mathematics, computer science, and engineering*, 2:257–397. www.math.univ-paris13.fr/~carlet/chap-fcts-Bool-corr.pdf.

Conway, J. H. and Sloane, N. J. A. (2013). *Sphere packings, lattices and groups*, volume 290. Springer Science & Business Media.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175. www.jmlr.org/papers/volume13/fortin12a/fortin12a.pdf.

GAP (2018). *GAP – Groups, Algorithms, and Programming, Version 4.8.10*. The GAP Group.

Lidl, R. and Niederreiter, H. (1994). *Introduction to finite fields and their applications*. Cambridge University Press.

Litzenberger, D. C. (2018). Pycrypto-the python cryptography toolkit. https://www.dlitz.net/software/pycrypto.

National Institute of Standards and Technology (2015). Secure hash standard(shs). Technical report. Federal Information Processing Standards Publication 180-4. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830. http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf.

pyca/cryptography Developers (2018). pyca/cryptography. https://cryptography.io/.

Sadohara, K. (2001). Learning of boolean functions using support vector machines. In *International Conference on Algorithmic Learning Theory*, pages 106–118. Springer. https://link.springer.com/chapter/10.1007/3-540-45583-3_10.

Siegenthaler, T. (1984). Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30(5):776–780. ieeexplore.ieee.org/document/1056949/.

The Python Packaging Authority (2017). Installation – pip 10.0.1. https://pip.pypa.io/en/stable/installing/ Accessed April 26, 2018.

The Sage Developers (2018). *SageMath, the Sage Mathematics Software System (Version 7.6)*.

Xiao, G.-Z. and Massey, J. L. (1988). A spectral characterization of correlation-immune combining functions. *IEEE Transactions on information theory*, 34(3):569–571. http://ieeexplore.ieee.org/document/6037/.

Appendix A. Code for Class Verification

```
from booleantools import BooleanFunction
import sys
import booleantools as bt
import itertools
import multiprocessing as mp
import json

# linear part + quadratic part
def analyze_polys(poly_gens, que, thread_no, n):
    for polys in poly_gens:
        for poly in polys:
            func = BooleanFunction(poly, n)
            if func.is_k_resilient(k=n-3) and not func.is_affine():
                que.put(poly)

    # Once all of the generators have executed
    que.put(thread_no)

def reduce_classes_dgen(class_list, new_f, n):
    """
    Produces a *MINIMALLY* reduced function list. This is by no means fully reduced.
    """
    for f in class_list:
        if new_f in f:
            return None

    class_list.append(get_class(new_f, n))
    return None

def reduce_classes(func_list):
    class_list = []
    for f in func_list:
        in_one = False
        for g in class_list:
            if f in g:
                in_one = True
        if in_one == False:
            class_list.append(f.get_orbit())
    return class_list

def get_class(f, n):
    perms = bt.Sym(n)
```

```
43      return [apply_permutation(f, sigma) for sigma in perms]
44
45  def apply_permutation(poly, perm):
46      def apply_perm_to_monomial(perm,monomial):
47          out = []
48          for var in monomial:
49              out.append(perm[var])
50          return out
51
52      out = [apply_perm_to_monomial(perm, i) for i in poly]
53      return out
54
55  def powerset(iterable):
56      s = list(iterable)
57      return list(itertools.chain.from_iterable(itertools.combinations(s, r) for
        r in range(len(s)+1)))
58
59  def func_generator(lin_part, non_lin_parts):
60      lin = [[mon] for mon in lin_part]
61      for nonlinear in non_lin_parts:
62          yield lin + list(nonlinear)
63
64
65  def main():
66      n = int(sys.argv[1])
67
68      nonlin = powerset(itertools.combinations(list(range(n)), 2))
69      nonlin.remove(())
70      lin = powerset(list(range(n)))
71
72      generator_list = [func_generator(linear, nonlin) for linear in lin]
73      size = len(generator_list)//16
74      chunked = [generator_list[i:i+size] for i in range(0, len(generator_list),
        size)]
75
76      que = mp.Queue()
77      threads = []
78      for generators in chunked:
79          thred = mp.Process(target=analyze_polys, args=(generators, que, len(
       threads), n))
80          thred.start()
81          threads.append(thred)
82
83      deadCnt = 0
84      f_out = []
85      while deadCnt < len(chunked):
86          f = que.get()
87          if isinstance(f, list):
88              reduce_classes_dgen(f_out, f, n)
89          else: # it's an int
90              deadCnt += 1
91              threads[f].join()
92
93      f_out = reduce_classes([BooleanFunction(f[0], n) for f in f_out])
```

```
94
95      with open("out/classes_%dv.json" % n, "w") as outfile:
96          outfile.write(json.dumps({i: f_out[i][0].listform for i in range(len(
        f_out))}, indent=4))
97
98  if __name__ == "__main__":
99      if(len(sys.argv) != 2):
100         print("USAGE: python3 generate_classes.py <n>")
101     else:
102         main()
```

Appendix B. Full Source Code for `booleantools` Package

```
1   import copy as _copy
2   import booleantools.fields as _fields
3   from itertools import combinations as _combs
4   from itertools import permutations as _perms
5
6   def Sym(n):
7       """
8       Creates a set containing all permutations in the symmetric group $S_n$.
9       Returns:
10          list: A set containing every permutation in $S_n$, in one-line
        notation.
11      """
12      return list(_perms([i for i in range(n)]))
13
14  GF2 = _fields.PrimeField(2)
15
16  class FieldFunction:
17      """
18      Represents a function over a Finite Field.
19      """
20
21      def __init__(self, listform, n, field):
22          self.listform = listform
23          self.n = n
24          self.field = field
25          self.__reduce()
26
27      def __call__(self, *args):
28          args = list(args)
29          if len(args) == 1 and hasattr(args[0], '__getitem__'):
30              args = args[0]
31          elif len(args) == 1 and isinstance(args[0], int):
32              args = _dec_to_base(args[0], self.n, self.field.order)
33
34          for pos, val in enumerate(args): #Simplification for inputs
35              args[pos] = self.field.get(val)
36
37          value = self.field.get(0)
38          for monomial in self.listform:
```

```
39              if monomial not in self.field:
40                  prod = self.field.get(1)
41                  for var in monomial:
42                      if var not in self.field:
43                          prod *= args[var]
44                      else:
45                          prod *= var
46                  value += prod
47              else:
48                  value += monomial
49
50          return self.field.value_of(value)
51
52
53      def apply_permutation(self, perm):
54          """
55          Applies a permutation to this function.
56          \\\[
57              f^\sigma(x)
58          \\\]
59
60          where sigma is in one line notation.
61
62          Args:
63              perm (list): The permutation to apply, in one line notation.
64
65          Returns:
66              FieldFunction: A function where the permutation was applied.
67          """
68          def apply_perm_to_monomial(perm,monomial):
69              out = []
70              for var in monomial:
71                  if var in self.field:
72                      out.append(var)
73                  else:
74                      out.append(perm[var])
75              return out
76
77
78          out = [apply_perm_to_monomial(perm, i) for i in self.listform]
79          return FieldFunction(out, self.n, self.field)
80
81      def __add__(a, b):
82          if a.field != b.field:
83              raise ValueError("Summands from different fields.")
84          if isinstance(b, _fields.PrimeField._FieldElement):
85              return FieldFunction(a.listform + [[b]], a.n, a.field)
86          else:
87              return FieldFunction(a.listform + b.listform, max(a.n, b.n), a.
    field)
88
89      def __mul__(a, b):
90          if a.field != b.field:
91              raise ValueError("Multiplicands are not from the same field.")
```

```
92
93          if isinstance(b, _fields.PrimeField._FieldElement):
94              out = [monomial + [b] for monomial in a.listform]
95              return FieldFunction(out, a.n, a.field)
96          else:
97              out = []
98              for monomial in a.listform:
99                  out += [monomial + monomial_b for monomial_b in b.listform]
100
101             return FieldFunction(out, max(a.n, b.n), a.field)
102
103
104     def tex_str(self, math_mode=False):
105         """
106         Creates a TeX String from this FieldFunction.
107         Args:
108             math_mode (bool, optional): Whether to return with surrounding '$
    '.
109         Returns:
110             str: A proper TeX String representing this function.
111         """
112         out = "" if not math_mode else "$"
113         flag = False
114         for monomial in self.listform:
115             out += " \\oplus " if flag else ""
116             for term in monomial:
117                 out += "x_{" + str(term) + "}"
118
119             flag = True
120
121         return out if not math_mode else out + "$"
122
123     def __str__(self):
124         return self.tex_str()
125
126     def __repr__(self):
127         return "FieldFunction(%s, %s, %s)" % (str(self.listform), str(self.n),
    \
128                 str(self.field))
129
130     def __reduce(self):
131         for i in range(len(self.listform)):
132             self.listform[i] = [val for val in self.listform[i] if val != self
    .field.get(1)]
133
134
135 class BooleanFunction(FieldFunction):
136     """
137     This class represents a boolean function ($\\mathbb{F}_2^n \\rightarrow
138     \\mathbb{F}_2) and implements a large amount of useful functions.
139     """
140     def __init__(self, listform, n):
141         """
142         Creates a boolean function on n variables.
```

```
143
144         Attributes:
145             listform (list): A list of the monomials this polynomial contains.
146                              Ex. \[x_1 \oplus x_2x_3\] is [[0], [1, 2]].
147             n (int): The number of variables, where n - 1 is the highest
148                      term in the list form.
149         """
150         super().__init__(listform, n, GF2)
151         _copyList = []
152
153         #This is done for space efficiency. Basically reduces coefficient mod
    2
154         for i in listform:
155             if i not in _copyList:
156                 _copyList.append(i)
157             else:
158                 _copyList.remove(i)
159
160         self.listform = _copyList
161         self.update_rule_table()
162
163     def hamming_weight(self):
164         """
165             Returns the Hamming Weight of this function.
166
167             Returns:
168                 int: The hamming weight of this function.
169         """
170         return sum(self.tableform)
171
172     def hamming_distance(self, other):
173         """
174         Determines the hamming distance of a function or a list of functions.
175
176         Args:
177             other (BooleanFunction): function or list of functions to find
    distance to.
178
179         Returns:
180             int: A list of distances if #other is a list, or a float if #other
     is another function.
181         """
182         if hasattr(other, "__getitem__"): #If other is a list
183             return [self.hamming_distance(f) for f in other]
184         else:
185             u = self.tableform
186             v = other.tableform
187             s = sum([_delta(u[k],v[k]) for k in range(len(u))])
188             return s
189
190     def walsh_transform(self):
191         """
192         Performs a Walsh transform on this function.
193         Returns:
```

```
194             list: A list containing the walsh transform of this function.
195         """
196         f = self.tableform
197         nbits = self.n
198         vecs = [(_dec_to_bin(x,nbits), x) for x in range(len(f))]
199         def Sf(w):
200             return sum([(-1)**(f[x]^_dot_product(vec,w)) for vec,x in vecs])
201         Sflist = [Sf(vec) for vec,x in vecs]
202         return Sflist
203 
204     def walsh_spectrum(self):
205         """
206         Generates the Walsh spectrum for this function.
207         Returns:
208             float: The Walsh spectrum of this function.
209         """
210         f = self.tableform
211         walsh_transform_f = self.walsh_transform()
212         spec = max([abs(v) for v in walsh_transform_f])
213         return spec
214 
215     def is_balanced(self):
216         """
217         Determines whether this function is balanced or not.
218 
219         # Returns
220             bool: True if balanced, False otherwise.
221         """
222         f = self.tableform
223         return sum(f) == len(f)/2
224 
225     def is_correlation_immune(self,k=1):
226         """
227         Determines if this function is k correlation immune.
228 
229         Args:
230             k (int): immunity level
231         """
232         if k > self.n:
233             raise BaseException("Correlation immunity level cannot be higher
    than the number of variables.")
234         f = self.tableform
235         walsh_transform = self.walsh_transform()
236         nbits = self.n
237         vectors_to_test = [_bin_to_dec(vec) for vec in
    weight_k_or_less_vectors(k,nbits)]
238         walsh_transform_at_weight_k = [walsh_transform[vec] for vec in
    vectors_to_test]
239         return walsh_transform_at_weight_k == [0]*len(
    walsh_transform_at_weight_k)
240 
241     def is_k_resilient(self,k=1):
242         """
243         Determines if this boolean function is k-resilient.
```

```
244
245         Args:
246             k (int): immunity level
247         """
248         return self.is_balanced() and self.is_correlation_immune(k=k)
249
250     def is_affine(self):
251         """
252         Determines if this function is affine.
253         Returns:
254             True if this function is affine, false otherwise.
255         """
256         return True if self.nonlinearity() == 0 else False
257
258     def get_orbit(self, perms=None):
259         """
260         Gets the orbit of this function under action of the symmetric group.
261         Args:
262             perms - default None. Uses this as a permutation set, otherwise
    the
263                     full symmetric group on n symbols.
264         Returns:
265             A list containing all functions in the orbit of this function.
266         """
267         return orbit_polynomial(self, perms)
268
269     def nonlinearity(self):
270         """
271         Gets the nonlinearity of this boolean function.
272
273         Returns:
274             int: Nonlinearity of this boolean function.
275
276         """
277         return int(2**(self.n-1) - 0.5*self.walsh_spectrum())
278
279     def linear_structures(self):
280         """
281             Creates a set of values that exist as linear structures of this
    polynomial.
282             Returns:
283                 set: Set of linear structures.
284         """
285         flatten = lambda l: [item for sublist in l for item in sublist]
286         linear_structs = set(flatten(self.listform))
287         for monomial in self.listform:
288             if len(monomial) > 1:
289                 linear_structs -= set(monomial)
290
291         return linear_structs
292
293     def apply_permutation(self, perm):
294         """
```

```
295         Applies a permutation to the ordering of the variables to this
    function.
296         Args:
297             perm - The permutation to apply.
298         Returns:
299             The newly permuted function.
300         """
301         f = super().apply_permutation(perm)
302         return BooleanFunction(f.listform, f.n)
303 
304     def __str__(self):
305         return self.tex_str()
306 
307     def __add__(a, b):
308         sum_f = FieldFunction.__add__(a, b)
309         return BooleanFunction(sum_f.listform, sum_f.n)
310 
311     def __mul__(a, b):
312         prod_f = FieldFunction.__mul__(a, b)
313         return BooleanFunction(prod_f.listform, prod_f.n)
314 
315     def __eq__(self,poly2):
316         return self.tableform == poly2.tableform
317 
318     def __repr__(self):
319         return "BooleanFunction(%s, %s)" % (str(self.listform), str(self.n))
320 
321     def update_rule_table(self):
322         rule_table_length = 2**self.n
323         rule_table = [0]*rule_table_length
324         for k in range(rule_table_length):
325             point_to_evaluate = _dec_to_bin(k, self.n)
326             rule_table[k] = self(*point_to_evaluate)
327         self.tableform = rule_table
328 
329     def __hash__(self):
330         return _bin_to_dec(self.tableform)
331 
332 
333 def getX(n, field=GF2):
334     """
335     Gets a list of all possible x_i in order, from 0 to n-1.
336     """
337     if field == GF2:
338         return [BooleanFunction([[i]], i+1) for i in range(0, n)]
339     else:
340         return [FieldFunction([[i]], i+1, field) for i in range(0, n)]
341 
342 def _gen_atomic(n, pos):
343         prod = BooleanFunction([[GF2.get(1)]], n)
344         for position, val in enumerate(_dec_to_bin(pos, n)):
345             if val == 1:
346                 f = BooleanFunction([[position]], n)
347                 prod *= f
```

```
348             else:
349                 f = BooleanFunction([[position], [GF2.get(1)]], n)
350                 prod *= f
351         if prod.tableform[pos] != 1:
352             raise BaseException("_gen_atomic failed! Please report on Github!")
353         return prod
354
355 def _GF2_to_ints(lst):
356     return [1 if x == GF2.get(1) else 0 for x in lst]
357
358
359 def generate_function(rule_no, n):
360     endFunc = BooleanFunction([], n)
361     binary_list = _dec_to_bin(rule_no, 2**n)
362     for pos, val in enumerate(binary_list[::-1]):
363         if val == 1:
364             endFunc += _gen_atomic(n, pos)
365
366     return endFunc
367
368 def _bin_to_dec(num):
369
370     """
371     Converts a binary vector to a decimal number.
372     """
373     return sum([num[i]*2**i for i in range(len(num))])
374
375 def _dec_to_bin(num,nbits):
376     """
377     Creates a binary vector of length nbits from a number.
378     """
379     new_num = num
380     bin = []
381     for j in range(nbits):
382         current_bin_mark = 2**(nbits-1-j)
383         if (new_num >= current_bin_mark):
384             bin.append(1)
385             new_num = new_num - current_bin_mark
386         else:
387             bin.append(0)
388     return bin
389
390 def _dec_to_base(num, nbits, base):
391     """
392     Creates a binary vector of length nbits from a number.
393     """
394     new_num = num
395     bin = []
396     for j in range(nbits):
397         current_bin_mark = base**(nbits-1-j)
398         if (new_num >= current_bin_mark):
399             bin.append(1)
400             new_num = new_num - current_bin_mark
401         else:
```

```
            bin.append(0)
    return bin

def _delta(x,y):
    """
    Returns 1 if x and y differ, 0 otherwise.
    """
    return x != y

def _hausdorff_distance_point(a,B):
    """
    Calculates the minimum distance between function a and the functions in
    the set B.
    """
    return min([a.hamming_distance(b) for b in B])

def _hausdorff_semidistance_set(A,B):
    return max([_hausdorff_distance_point(a,B) for a in A])

def hausdorff_distance(X,Y):
    """
    Calculates the Hausdorff distance between two sets of boolean functions.
    """
    HD1 = _hausdorff_semidistance_set(X,Y)
    HD2 = _hausdorff_semidistance_set(Y,X)
    return max([HD1,HD2])

def _dot_product(u,v):
    """
    Basic mod 2 dot product.
    """
    s = sum(u[k]*v[k] for k in range(len(u)))
    return s%2

def weight_k_vectors(k,nbits):
    """
    Generates all vectors with hamming weight k.
    """
    nums = range(nbits)
    vector_set_to_return = []
    k_combinations = [list(x) for x in _combs(nums,k)]
    for j in k_combinations:
        vec_to_add = [int(y in j) for y in range(nbits)]
        vector_set_to_return.append(vec_to_add)
    return vector_set_to_return

def weight_k_or_less_vectors(k, nbits):
    """
    Generates all vectors of weight k on nbits bits.
    Args:
        k - weight
        nbits - the number of bits
    Returns:
        All vectors of weight k on nbits bits.
```

```
455     """
456     output = []
457     for i in range(0, k+1):
458         output += weight_k_vectors(i, nbits)
459 
460     return output
461 
462 def _product(x):
463     return reduce((lambda y,z : y*z), x)
464 
465 def duplicate_free_list_polynomials(list_of_polys):
466     """
467     Takes a list of boolean functions and generates a duplicate free list of
    polynomials.
468 
469     # Arguments
470         list_of_polys (BooleanFunction): A list of polynomials.
471 
472     # Returns
473         list: A duplicate free list of functions
474     """
475     outlist = []
476     for poly in list_of_polys:
477         if True not in [poly == poly_in_out for poly_in_out in outlist]:
478             outlist.append(poly)
479     return outlist
480 
481 def orbit_polynomial(polynomial, permset=None):
482     """
483         Orbits a polynomial using the given permutation set.
484 
485         Args:
486             permset: A set of permutations to apply to the function
487         Returns:
488             A list of the polynomials created by the given orbits.
489     """
490     if permset is None:
491         permset = Sym(polynomial.n)
492     return duplicate_free_list_polynomials([polynomial.apply_permutation(i)
    for i in permset])
493 
494 def orbit_polynomial_list(polynomial_list, permset=None):
495     """
496     Orbits a list of polynomials using the given permutation set.
497 
498     Returns:
499         A list of lists of the polynomials created by the given orbits.
500     """
501     return [orbit_polynomial(polynomial, permset) for polynomial in
    polynomial_list]
502 
503 
504 def siegenthaler_combination(f1,f2,new_var):
505     """
```

```
506     Generates a Siegenthaler Combination of two functions.
507 
508     Args:
509         f1 (BooleanFunction): The first function
510         f2 (BooleanFunction): The second function
511         new_var (int): New variable for the combined function.
512 
513     Returns:
514         The Siegenthaler combination of $f_1$ and $f_2$
515 
516     """
517     f1_times_new_var = f1 * new_var
518     f2_times_one = f2
519     f2_times_new_var = f2 * new_var
520     return f1_times_new_var + f2_times_one + f2_times_new_var
521 
522 def generate_all_siegenthaler_combinations(func_list,new_var):
523     """
524     Generates all of the possible Siegenthaler combinations
525     of the given functions.
526 
527     Args:
528         func_list - A list of functions to perform the Siegenthaler
    combination function on.
529 
530     Returns:
531         A list of all possible Siegenthaler combinations for the given
    functions.
532     """
533     all_siegenthaler_combinations = []
534     for f1 in func_list:
535         for f2 in func_list:
536             f1f2siegenthalercombination = siegenthaler_combination(f1,f2,
    new_var)
537             all_siegenthaler_combinations.append(f1f2siegenthalercombination)
538     return all_siegenthaler_combinations
539 
540 def min_nonzero_dist(poly1, classA):
541     """
542     Determines the minimum nonzero distance between a polynomial and its
    nearest neighbor.
543 
544     Args:
545         poly1 - A boolean function
546         classA - A class of boolean functions.
547 
548     Returns:
549         The minimum nonzero distance between poly1 and every element of classA
    .
550     """
551     dists = [poly1.hamming_distance(f) for f in classA]
552     min_nonzero = float("inf")
553     for dist in dists:
554         if dist != 0 and dist < min_nonzero:
```

```
555             min_nonzero = dist
556
557     return dist
558
559 def reduce_to_orbits(f_list, permset):
560     """
561     Reduces a list of functions to a list of function classes given a
        permutation set.
562     """
563     basic_polys = []
564     flatten = lambda l: [item for sublist in l for item in sublist]
565     for f in f_list:
566         if f not in flatten([orbit_polynomial(permset, basic) for basic in
        basic_polys]):
567             basic_polys.append(f)
568     return basic_polys
```

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE, WESTERN CAROLINA UNIVERSITY, CULLOWHEE, NC 28723
E-mail address: adpenland@email.wcu.edu

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE, WESTERN CAROLINA UNIVERSITY, CULLOWHEE, NC 28723
E-mail address: wsrogers3@catamount.wcu.edu